```
-- 1.
-- a.

scale_nums :: [Integer] -> Integer -> [Integer]
-- scale_nums xs x = map (\x -> x*) xs
scale_nums xs s = map (\x -> x*s) xs
-- the x * is actually a function being passed to the map function

-- b.
-- only_odds :: [[a]] -> [[a]]
only_odds :: [[Integer]] -> [[Integer]]
only_odds = filter (\lst -> all odd lst)
-- here all is a function that takes a function and a list and returns a boolean and so if the lst
satisfies the function then it returns true and the filter function will return the list

-- c.
largest :: String -> String -> String
largest first second =
   if length first >= length second then first else second

largest_in_list :: [String] -> String
largest_in_list lst = foldl (\acc x -> largest acc x) "" lst
-- have the accumulator come before the x because the largest function takes in two strings and
returns the largest one and so the accumulator is the first string and the x is the second string
```

```haskell
-- 2.
-- a.
count_if :: (a -> Bool) -> [a] -> Integer
count_if _ [] = 0
count_if func (x:xs)
   | func x = 1 + count_if func xs -- if the first element satisfied the function
   | otherwise = count_if func xs -- if the first element did not satisfy the function

-- b.
count_if_with_filter :: (a -> Bool) -> [a] -> Int
count_if_with_filter func lst = length (filter func lst)

-- c.
count_if_with_fold :: (a -> Bool) -> [a] -> Int
-- count_if_with_fold func lst = foldl (\acc x -> if func x then acc + 1 else acc) 0 lst
count_if_with_fold p xs = foldl (\count element -> if p element then count + 1 else count) 0 xs
```

-- 3.
-- a.
{- Currying and partial application are interconnected but opposite. Currying is the process of taking a function with multiple arguments
and turning it into a series of nested functions which each take a single argument. Meanwhile Partial Function Application is when we only
partially pass in the arguments to a multi-parametered function. Therefore, it leaves us with a new function that can take in the remaining
parameters to be executed. -}

-- b.
{-
Due to currying, a->(b->c) is equivalent to a->b->c. This is because both functions ultimately take in a function and then return a function.
The second expression essentially takes in the argument of type a and then returns a function that takes in an argument of type b and that returns type c.
Both statements are equivalent due to currying, as the functions can be partially applied one by one.
-}

-- c. foo x y z t = map t [x,x+z..y]
foo :: Integer -> Integer -> Integer -> (Integer -> a) -> [a]
foo = \x -> \y -> \z -> \t -> map t [x,x+z..y]

-- 4.
-- a.
-- The variable a was captured in this statement
-- b.
-- The variable b was captured in this statemnent
-- c.
-- The variable f, c, d and e was captured in this statement
-- d.
-- The 4 would be bound to a
-- The 5 would be bound to b
-- c is bound to the scope of '4' as it is within the parameter of a
-- d is bound to the scope of '5' as it is within the parameter of b
-- e is bound to the scope of '6'
-- f is bound to the scope of '7'

-- 5.
-- Haskell closures are also first-class citizens like function pointers.
-- However, closures additionally have the option to compute upon anonymous functions
-- due to the lambda structure, where as C functions are required to be defined.
-- Arguments are able to binded to a function in a closure
-- where as the same cannot be achieved in a C pointer.

```
-- 6.
-- a.
-- data InstagramUser = Influencer | Normie deriving (Eq, Show)

-- -- b.
-- lit_collab :: InstagramUser -> InstagramUser -> Bool
-- lit_collab x y =
--     if (x == Influencer && y == Influencer) then True
--                                             else False


-- c.
-- data InstagramUser = Influencer [String] | Normies deriving (Eq, Show)

-- d.
-- is_sponsor :: InstagramUser -> Bool
-- is_sponsor = \x -> case x of
--          Influencer _ -> True
--          Normies -> False


-- e.
data InstagramUser = Influencer [String] [InstagramUser] | Normies deriving (Eq, Show)


-- f. *** NOT DONE YET
-- count_influencers :: InstagramUser -> Integer
-- count_influencers = \x -> case x of
--                  Normies -> 0
--                  InstagramUser x:xs -> 1 + count_influencers xs
count_influencers :: InstagramUser -> Integer
count_influencers Normies = 0 -- People who are normies have no followers
count_influencers (Influencer _ followers) =
                        1 + sum (map count_influencers followers)


-- g.
-- After running :t Influencer, the output is as follows: Influencer :: [String] -> [InstagramUser] ->
InstagramUser
-- we can infer that the influencer constructor accepts two arguments a list of strings and a list of
instagramUsers that represents its followers
```

```
-- 7.
-- a.
-- data LinkedList = EmptyList | ListNode Integer LinkedList deriving Show
-- ll_contains :: LinkedList -> Integer -> Bool

-- ll_contains EmptyList _ = False
-- ll_contains (ListNode val rLst) target
--        | val == target = True
--        | otherwise = ll_contains rLst target




-- b.
data LinkedList = EmptyList | ListNode Integer LinkedList deriving Show
{-
Have an integer for indexing and one for the actual value?
This will allow the traversal to match to the indexing index and then insert at the desired
zero-based index.
-}

-- c.
ll_insert :: LinkedList -> Integer -> Integer -> LinkedList
-- when inserting a new node, insert the actual node not the entire linked list
ll_insert EmptyList val _ = ListNode val EmptyList -- Insert at the beginning (for an empty list or
negative index)
ll_insert (ListNode curr_val rLst) desired_val index
   | index <= 0 = ListNode desired_val (ListNode curr_val rLst) -- Insert at the beginning (for a
positive index)
   | otherwise = ListNode curr_val (ll_insert rLst desired_val (index - 1)) -- Recursively insert at
the specified index
```

```haskell
-- 8.
-- a.
-- int longest_run(const std::vector<bool>& v) {
--     int maximumRun = 0
--     int currentRun = 0

--     for (bool value : vec) {
--         if (value) { currentRun++ }
--         else { currentRun = 0 }
--         if (currentRun > maximumRun) {
--             maximumRun = currentRun
--         }
--     }
--     return maximumRun

-- }

-- b.
-- longest_run :: [Bool] -> Int
-- longest_run
-- Using Haskell, write a function named longest_run that takes in a list of Bools and returns the
length of the longest consecutive sequence of True values in that list.
longest_run :: [Bool] -> Int
longest_run = go 0 0
    where
        go maxRun currentRun [] = maxRun
        go maxRun currentRun (x:xs)
            | x == True = go (max maxRun (currentRun + 1)) (currentRun + 1) xs
            | otherwise = go maxRun 0 xs


-- d.
data Tree = Empty | Node Int [Tree] deriving Show
max_tree_value :: Tree -> Int
max_tree_value Empty = 0
max_tree_value (Node val []) = val
max_tree_value (Node val rTree) = max val (maximum (map max_tree_value rTree))
```

```
-- 9.
fibonnaci :: Int -> [Int]
-- fibonnaci 0 lst = 1 : lst
-- fibonnaci 1 lst = 1 : lst
-- fibonnaci num lst
--     | fibonnaci (num-1) lst + fibonnaci (num-2) lst = fibonnaci num lst : lst
fibonnaci n
    | n <= 0 = [] -- return an empty list
    | otherwise = take n (fib 1 1)
  where
    fib a b = a : fib b (a + b)
-- building the fibonnaci sequence recursively but from the bottom up
-- That way the sequence is in order
```